Job-Level Proof Number Search

I.-Chen Wu, *Member, IEEE*, Hung-Hsuan Lin, Der-Johng Sun, Kuo-Yuan Kao, Ping-Hung Lin, Yi-Chih Chan, and Po-Ting Chen

***Abstract*—This paper introduces an approach, called generic job-level search, to leverage the game-playing programs which are already written and encapsulated as jobs. Such an approach is well suited to a distributed computing environment, since these jobs are allowed to be run by remote processors independently. In this paper, we present and focus on a job-level proof number search (JL–PNS), a kind of generic job-level search for solving computer game search problems, and apply JL–PNS to solving automatically several *Connect6* positions, including some difficult openings. This paper also proposes a method of postponed sibling generation to generate nodes smoothly, and some policies, such as virtual win, virtual loss, virtual equivalence, flagging, or hybrids of the above, to expand the nodes. Our experiment compared these policies, and the results showed that the virtual-equivalence policy, together with flagging, performed the best against other policies. In addition, the results also showed that the speedups for solving these positions are 8.58 on average on 16 cores.**

***Index Terms*—*Connect6*, desktop grids, job-level proof number search (JL–PNS), proof number search, threat-space search.**

I. Introduction

PROOF NUMBER SEARCH (PNS), proposed by Allis *et al.* [1], [3], is a kind of best-first search algorithm that was successfully used to prove or solve theoretical values [12] of game positions for many games [1]–[3], [11], [21]–[23], [28], such as *Connect-Four*, *Gomoku*, *Renju*, *Checkers*, *Lines of Action*, *Go*, and *Shogi*. Like most best-first searches, PNS has a well-known disadvantage: the requirement of maintaining the whole search tree in memory. As a result, many variations have been proposed to avoid this problem, such as PN^2 [5], DFPN [14], [18], [19], PN* [23], PDS [28], and parallel PNS [13], [21]. For example, PN^2 used two-level PNS to reduce the size of the maintained search tree.

In this paper, we introduce a new approach, named generic job-level search, where a search tree is maintained by a process, also called the client in this paper. Search tree nodes are evaluated or expanded/generated by leveraging the game-playing programs which are already well written and encapsulated as jobs, usually heavyweight jobs requiring tens of seconds or more. Such an approach is well suited to a distributed computing environment, since these jobs are allowed to be run independently by remote processors.

In this paper, we present and focus on a job-level proof number search (JL–PNS), a kind of generic job-level search. We demonstrate JL–PNS by applying JL–PNS to solving automatically several *Connect6* positions, including openings. We use NCTU6, a *Connect6* program, as the job. NCTU6 has won the gold medal at *Connect6* tournaments [16], [26], [31], [34], [35], [37] several times since 2006, and has defeated many top level human *Connect6* players [17] in man–machine *Connect6* championships since 2008.

The generic job-level search approach as well as JL–PNS has the following advantages.

- It develops jobs (usually heavyweight jobs or well-written programs) and the JL–PNS independently, except for a few extra processes required to support JL–PNS from these jobs. As described in this paper, these processes are relatively low level.
- It dispatches jobs to remote processors in parallel. JL–PNS is well suited to parallel processing, as mentioned above.
- It maintains the JL–PNS tree inside the memory of clients without much problem. Since well-written game-playing programs normally support accurate domain-specific knowledge to a certain extent, the search trees require fewer nodes to solve the game positions (when compared with a pure PNS program using one process only). In our experiments for *Connect6*, the search tree usually contains no more than one million nodes, which fits well into (client) process memory. For example, assume that it takes one minute to run a job (to generate one node). A parallel system with 60 processors takes about 11 days to build a tree of up to one million nodes. Should we need to run many more than one million nodes, we can split the JL–PNS tree into several nodes, each per client.
- It easily monitors the search tree. Since the maintenance cost for the JL–PNS tree is low, the client that maintains the JL–PNS tree can support more graphical user interface (GUI) utilities to let users easily monitor the running of the whole JL–PNS tree in real time. In fact, our JL–PNS client is embedded into a game record editor environment. An extra benefit of this is to allow users or experts to look into the search tree during the running time, and to help choose the best move to search in the case that the program does not find the best move to win (see [30] and [35]).

Manuscript received April 21, 2012; revised July 27, 2012; accepted October 02, 2012. Date of publication October 12, 2012; date of current version March 13, 2013. This work was supported in part by the National Science Council of the Republic of China (Taiwan) under Contracts NSC 97-2221-E-009-126-MY3, NSC 99-2221-E-009-102-MY3, and NSC 99-2221-E-009-104-MY3.

I.-C. Wu, H.-H. Lin, D.-J. Sun, Y.-C. Chan, and P.-T. Chen are with the Department of Computer Science, National Chiao Tung University, Hsinchu 30050, Taiwan (e-mail: icwu@csie.nctu.edu.tw).

K.-Y. Kao is with the Department of Information Management, National Penghu University, Penghu 880, Taiwan.

P.-H. Lin is with the Department of Computer Science, National Chiao Tung University, Hsinchu 30050, Taiwan and also with the Information and Communications Research Laboratories, Industrial Technology Research Institute, Hsinchu 31040, Taiwan.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCIAIG.2012.2224659

1943-068X/$31.00 © 2012 IEEE

For node generation of JL–PNS, we need to select nodes and then expand them. For node expansion, this paper proposes a method, named postponed sibling generation method, to help expand the selected nodes.

In the preliminary version [31] of this paper, for node selection, we used the virtual-win/virtual-loss (VW/VL) policies which assume a win/loss for the node being selected for expansion. Since our proposal in [31], JL–PNS has also been applied to solving the game *Breakthrough* with 6 × 5 boards in [20]. Saffidine *et al.* [20] used JL–PNS based on a PN^2 search tree, and used a flag mechanism to help select the next node to run, instead of VW/VL policies.

This paper proposes a new policy, named virtual equivalence (VE). In this policy, it is assumed that the value of a game position is close to (or equal to) that of the position for the best move, and that the value for the nth best move is close to (or equal to) that for the $(n+1)$th best move. We also propose some variants of VE. Our experiments showed that one of the VE variants performed the best and improved the VW/VL policies by a factor of about 1.86.

Using JL–PNS with the job, NCTU6, on desktop grids (a kind of volunteer computing system[1] [4], [9], [24], [30]), we solved several *Connect6* positions including several difficult three-move openings, as shown in Fig. 12. For some of these openings, none of the human *Connect6* experts had been able to find the winning strategies. These solved openings include the popular Mickey-Mouse opening,[2] [25], as shown in Fig. 12(i).

This paper is organized as follows. Section II defines job-level computation. Section III reviews PNS, *Connect6*, and program NCTU6. Section IV describes our J–LPNS. Section V presents experiments for JL–PNS. Section VI discusses some related work and some miscellaneous issues for JL–PNS, such as the overhead. Section VII provides concluding remarks.

II. Job-Level Computation

This section introduces our job-level computation for computer games applications. The job-level computation model is proposed in Section II-A. The generic search is described in Section II-B, while the generic job-level search is described in Section II-C.

A. Job-Level Computation Model

In the model, the computation is done by a client, which dynamically creates jobs to do. For example, in a computer game application, the client creates one job for each move in a position, and each job evaluates the value of the corresponding move.

A job-level system, as shown in Fig. 1, includes a set of workers, which helps perform jobs. In the system, jobs created by clients are dispatched to a broker, which selects available workers to perform.

As shown in Fig. 2, messages between the client and the job-level system mainly include the following three things: job submission, notification of an idling worker, and the job result. The first one is from the client to the system, while the next two are the other way around.

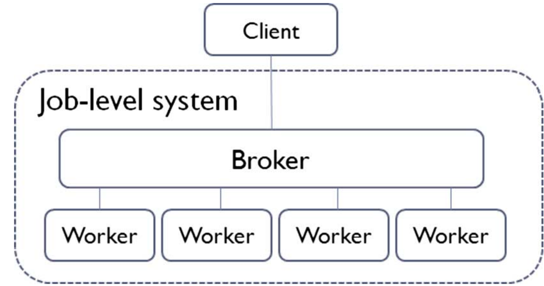


Fig. 1. The job-level computation model.

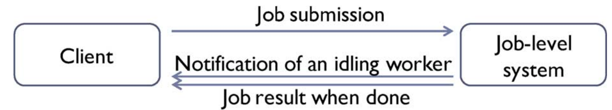


Fig. 2. The messages between a client and the job-level system.

In the job-level model, the clients wait passively for the available workers to submit jobs. Whenever a worker is available for computing a job, it will notify the broker, and the broker will in turn notify the client that one worker is available. Then, the client submits one job, if any, to the broker, which in turn dispatches the job to the worker. When completing the job, the worker sends the job result back to the client, which then updates according to the result. During the update, more jobs may be generated for job dispatching.

In the model, the client usually does not actively submit a large number of jobs to the job-level system in advance. For example, for a position with ten moves, assume that the client actively creates ten jobs each per move in advance, and submits them to a job-level system with two workers only. In the case that one of these moves turns promising, say nearly winning, a good strategy is to shift the computing resources from other moves temporarily to this promising move and its descendants. However, in the case of submitting a large number of jobs in advance, the workers still work on other moves, unrelated to the promising move.

The job-level computation model was realized in a desktop system designed by Wu *et al.* [30]. In practice, a job-level system may also support some other messages, such as abortion messages, ask-info messages, etc. Abortion messages can be used to abort running jobs, which are no longer interesting. For example, if a move is found to be a sure win from a job, other jobs for its sibling moves are no longer interesting and, therefore, can be aborted immediately. Ask-info messages can be used to ask the job-level system to report the job status for monitoring. The details [30] are omitted in this paper.

B. Generic Search

In this section, we describe generic best-first search, or simply called generic search, that fits many search techniques, like PNS and Monte Carlo tree search (MCTS) [6]. Generic search is associated with a search tree, where each node represents a game

[1]A desktop grid is developed for volunteer computing, which aimed to harvest idle computing resources for speeding up high throughput. It is a kind of distributed computing.

[2]The opening was called this by *Connect6* players because White 2 and Black 1 together look like the face of Mickey Mouse to them.

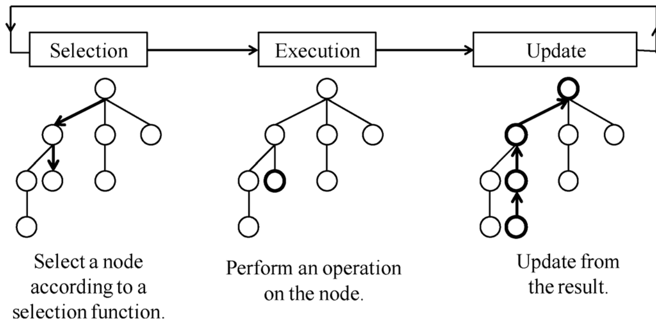


Fig. 3. Outline of a job-level computation model for a single core.

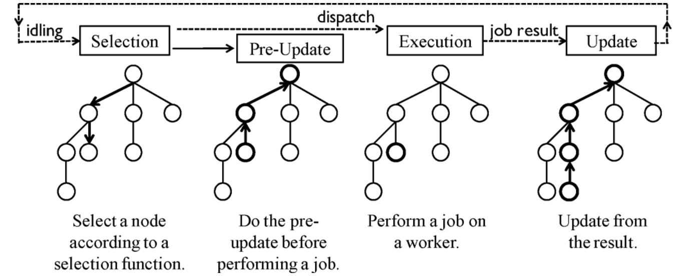


Fig. 4. Outline of a job-level computation model.

position. The process of a generic search usually repeats the following three phases: selection, execution, and update, as shown in Fig. 3.

First, in the selection phase, a node is selected according to a selection function based on some search technique. For example, PNS selects the most proving node (MPN, which will be described in more detail in Section III-A); and MCTS selects a node based on the so-called tree policy (defined in [6] and [10]). Note that the search tree is supposed to be unchanged in this phase.

Second, in the execution phase, operation $J(n)$ is performed on the selected node n, but does not change the search tree yet. For example, find the best move from node n, expand all moves of n, or run a simulation from n for MCTS. After performing job $J(n)$, a result is obtained. For the above example, the result is the best move, all the expanded moves, or the result of a simulation, respectively.

Third, in the update phase, the search tree is updated according to the job result. For the above example, a node is generated for the best move, nodes are generated for all expanded moves, and the status is updated on the path to the root.

C. Generic Job-Level Search

From Section II-B, operation $J(n)$ on the selected node n does not change the search tree. Therefore, operation $J(n)$ can be done as a job by another worker remotely in a job-level system. The job submission may include some data required by $J(n)$, such as the neighboring nodes or the path to the root. Thus, a generic search becomes a generic job-level search, run in a job-level computation model with one worker only.

However, since a generic search repeats the three phases sequentially (as shown in Fig. 3), the job-level system with multiple workers is not efficient. Thus, in generic job-level search, the computation model is changed to be run in parallel, as shown in Fig. 4. The details are described as follows.

As described in Section II-A, the client waits passively for notification of idling workers. When receiving a notification, the client selects a node in the selection phase and then dispatches a job, if any, to the worker for execution. When the job is done, the worker sends the result back to the client. When receiving the result, the client runs the third phase to update the search tree from the result. These are all performed in an event-driven model.

In a job-level system with multiple workers, one issue is that in the above model the client will select the same node for multiple notifications of idling workers, if no other results are found and used to update the search tree in the interim.

In order to solve this issue, we modify the model as in Fig. 4 by adding one phase, called the pre-update phase, after the selection phase and before job submission. In this phase, several policies can be used to update the search tree. For example, the flag policy sets a flag on the selected node, so that the flagged nodes will not be selected again.

Another issue deals with growth of the search tree, such as node expansion or generation from the search tree. Consider a case that a leaf node n is selected. If job $J(n)$ is to expand all moves, all the child nodes (corresponding to these moves) are expanded from n. However, in many cases, it is inefficient to expand all moves in the job-level model (as described in Section III-B).

If $J(n)$ is to find the *best* move, then, in the update phase, the node corresponding to the best move should be generated (usually by running a game-playing program for $J(n)$, such as NCTU6). However, the question is when and how to expand other nodes such as those for the second best node from n, the third best, etc. For this problem, we propose a more general job $J(n, C(n))$, which finds the best move among all the moves excluding those in list $C(n)$, where $C(n)$ is a list of prohibited moves. Thus, for n, we can use $J(n, \emptyset)$ to find the best move n_1, and then use $J(n, \{n_1\})$ to find the second best move n_2, and so on.

III. Background

This paper demonstrates the job-level model by using job-level proof number search to solve some openings of *Connect6* based on a *Connect6* program NCTU6. PNS is reviewed in Section III-A, and *Connect6* and NCTU6 are described in Section III-B.

A. Proof Number Search

For simplicity of discussion about PNS, we follow in principle the definitions and algorithms in [1] and [3]. PNS is based on an AND/OR search tree where each node n is associated with proof/disproof numbers $p(n)$ and $d(n)$, which represent the minimum numbers of nodes to be expanded to prove/disprove n. Basically, all leaves' $p(n)/d(n)$ are initialized to 1/1. Values $p(n)/d(n)$ are $0/\infty$ if the node n is proved, and $\infty/0$ if it is disproved. PNS repeatedly chooses a leaf called MPN to expand, until the root is proved or disproved. The details of

choosing MPN and maintaining the proof/disproof numbers can be found in [1] and [3] and, therefore, are omitted in this paper.

An important property related to MPN is: if the selected MPN is proved (disproved), the proof (disproof) number of the search tree decreases by one. The property, called MPN property in this paper, can be generalized as follows.

- If the selected MPN is proved (disproved), the proof (disproof) number of the node, whose subtree includes the MPN, decreases by one, and the disproof (proof) number of it remains the same or increases.

B. Connect6 and NCTU6

Connect6 is a kind of six-in-a-row game that was introduced by Wu *et al.* [32], [33]. Two players, named Black and White in this paper, alternately play one move by placing two black and white stones, respectively, on empty intersections of a *Go* board (a 19 × 19 board) in each turn. Black plays first and places one stone initially. The winner is the first to get six consecutive stones of his own horizontally, vertically, or diagonally.

NCTU6 is a *Connect6* program, developed by a team led by I.-C. Wu, as also described in Section I. In this section, we review the results from [27] and [35] as follows. NCTU6 included a solver that was able to find victory by continuous four (VCF), a common term for winning strategies in the *Renju* community. More specifically, VCF for *Connect6*, also called VCST, wins by making continuous moves with at least one four (a threat which causes the opponent to defend) and ends with connecting up to six in all subsequent variations.

From the viewpoint of lambda search, VCF or VCST is a winning strategy in the second order of threats, according to the definition in [35], that is, a Λ_a^2-tree (similar to a λ_a^2-tree in [27]) with value 1. Lambda search, as defined by Thomsen [27], is a kind of threat-based search method, formalized to express different orders of threats. Wu and Lin [35] modified the definition to fit *Connect6* as well as a family of k-in-a-row games and changed the notation from λ_a^i to Λ_a^i.

NCTU6-verifier (Verifier) is a verifier modified from NCTU6 by incorporating a lambda-based threat-space search, and used to verify whether the player-to-move loses in the position, or to list all the defensive moves that may prevent the player from losing in the order Λ_a^2. If no moves are listed from a position, Verifier is able to prove that the position is a loss. If some moves are listed, Verifier is able to prove that those not listed are losses. In some extreme cases, Verifier may report up to tens of thousands of moves.

One issue for *Connect6* is that the game lacks openings for players, since the game is still young when compared with other games such as *Chess*, *Chinese Chess*, and *Go*. Hence, it is important for the *Connect6* player community to investigate more openings quickly. For this issue, Wu *et al.* [30] designed a desktop grid, such as the job-level system, to help human experts build and solve openings.

In the earliest version of the grid, both NCTU6 and Verifier were the two jobs used, and a game record editor environment was utilized to allow users to select and dispatch jobs to free workers. NCTU6 was used to find the best move from the current game position, while Verifier was used to expand all the nodes (namely for all the defensive moves). This environment helped human experts build and solve openings manually.

In this paper, the system is modified to support a job-level system where JL–PNS can be used to create and perform jobs automatically. Both NCTU6 and Verifier are supported as jobs. NCTU6 jobs take tens of seconds on the average (statistics are given in Section V), and Verifier jobs take a wide variety of times, from one minute up to one day, depending on the number of defensive moves. As above, in some extreme cases, Verifier may generate a large number of moves in JL–PNS, which is resource consuming for both computation and memory resources. Thus, Verifier is less feasible in practice.

In order to solve this problem, we modify NCTU6 to support the following two additional functionalities.

1) Support $J(n, C(n))$. Given position n and a list of prohibited moves $C(n)$ as input, NCTU6 generates the best move among all the moves outside the list. As described in Section II-C, this can be used to find the best move of a position, the second best, etc.
2) For each job $J(n, C(n))$, report a sure loss in the job result, if none of the nonprohibited moves can prevent a loss.

Supporting the first functionality, we can use the modified NCTU6 to find the best move of a position, the second best, etc., as described in Section II-C. Supporting the second functionality, we can expand all the moves like Verifier. Thus, NCTU6 is able to replace Verifier with JL–PNS.

IV. JOB-LEVEL PROOF NUMBER SEARCH

This section presents JL–PNS and demonstrates it by using NCTU6, a *Connect6* program, to solve *Connect6* positions automatically. JL–PNS uses PNS (described in Section III-A) to maintain a search tree in the client, and runs in four phases following the generic job-level search, described in Section II-C.

In the selection phase, MPN is selected, and jobs are created from MPN for execution on workers in the execution phase. In Section IV-B, we propose a method, called postponed sibling generation, to create jobs. In the update phase, the move in the job result is used to generate the corresponding new node, and the evaluated value of the move is used to initialize the proof/disproof numbers of the node and to update others in the search tree, described in Section IV-A. In the pre-update phase, several policies are proposed and described in Section IV-C.

A. Proof/Disproof Number Initialization

This section briefly describes how to apply the domain knowledge given by NCTU6 to initialization of the proof/disproof numbers. Since it normally takes tens of seconds to execute an NCTU6 job, it becomes critical to choose carefully a good MPN to expand, especially when there are many candidates with 1/1 as the standard initialization. In [1], Allis suggested several methods, such as the use of the number of nodes to be expanded, the number of moves to the end of games, or the depth of a node.

Our approach is simply to trust NCTU6 and use its evaluations on nodes (positions) to initialize the proof/disproof numbers in JL–PNS, as shown in Table I. The status Bw indicates

TABLE I
GAME STATUS AND THE CORRESPONDING INITIALIZATIONS

Status	Bw	B4	B3	B2	B1	stable	unstable2
p(n)/d(n)	0/∞	1/18	2/12	3/10	4/8	6/6	4/4
Status	Ww	W4	W3	W2	W1	unstable1	
p(n)/d(n)	∞/0	18/1	12/2	10/3	8/4	5/5	

that Black has a sure win, so the proof/disproof numbers of a node with Bw are $0/\infty$. For simplicity of discussion, this paper looks to prove a game when Black wins, unless explicitly specified. Statuses B1–B4 indicate that the game favors Black with different levels of win probability, where B1 indicates to favor Black with the least probability and B4 with the most (i.e., Black has a very good chance of a win in B4) according to the evaluation by NCTU6. Similarly, statuses W* are for White. The status "stable" indicates that the game is stable for both players, while both "unstable1" and "unstable2" indicate unstable, where unstable2 is more unstable than unstable1. Proof/disproof numbers of these unstable statuses are smaller than those of "stable," since it is assumed to be more likely to prove or disprove "unstable" positions.

Of course, there are many different kinds of initializations other than those in Table I. Our philosophy is simply to pass the domain-specific knowledge from NCTU6 to JL–PNS. Different programs or games naturally have different policies on initializations from practical experiences.

B. Postponed Sibling Generation

In this section, we describe how to create jobs after an MPN n is selected. Straightforwardly from PNS, node n is expanded and all of its children are generated. Unfortunately, in *Connect6*, the number of children is up to tens of thousands of nodes usually. If we use Verifier to help remove some losing moves, it may still take a huge amount of computation time, as described in Section III-B. Thus, it becomes more efficient and effective to generate a node at a time. However, in PNS, the MPN is a leaf in the search tree. If we always generate the best move from the MPN, then there are no choices to generate the second best move, the third best, etc. In order to solve this problem, we propose a method called postponed sibling generation as follows.

- Assume that for node n, the ith move n_i is already generated, but $(i+1)n_{i+1}$ is not yet. When node n_i is chosen as the MPN for expansion, generate the best move of n_i by $J(n_i, \emptyset)$ and generate n_{i+1} by $J(n, \{n_1, \ldots n_i)$ simultaneously. The example in Fig. 5 illustrates this. Assume that node n_3 is chosen as the MPN. Then, generate the best move of n_3 by $J(n_3, \emptyset)$ and generate n_4 by $J(n, \{n_1, n_2, n_3\})$ simultaneously. On the other hand, if branch n_1 or n_2 is chosen, do not generate n_4 as of yet.
- In an attacker-to-move node, assume that a generated move is reported to be a sure loss to the attacker. Then, generate no more moves from the node, since others are also sure losses as per the second functionality described in Section III-B. For example, in Fig. 5, assume that $J(n, \{n_1, n_2, n_3\})$ reports a sure loss when generating n_4. From the second functionality, all the moves except for n_1, n_2, and n_3 are sure losses. Thus, it is no longer necessary to expand node n. In this case, all children of n are generated, and n_4 behaves as a *stopper*. Note that it is similar in the case that node n is an AND node.

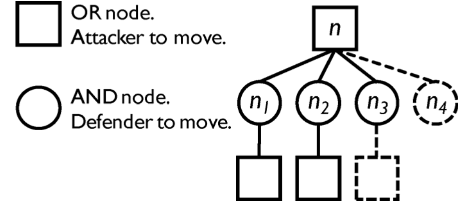


Fig. 5. Expanding n_3 and n (to generate n_4) simultaneously.

Since NCTU6 supports $J(n, C(n))$ and is able to report a sure loss, as described in Section III-B, NCTU6 can support postponed sibling generation.

As shown in Fig. 5, postponed sibling generation fits parallelism well, since generating n_4 and expanding n_3 can be both performed simultaneously. Some further issues are described as follows.

One may ask: what if we choose to generate n_4 before expanding n_3? Assume that one player, say the attacker, is to move in the OR node n. As per the first additional functionality described in Section III-B, move n_3 is assumed to be better for the attacker than n_4, according to the evaluation of NCTU6. In this case, thecondition $p(n_3) \leq p(n_4)$ holds. Thus, node n_3 must be chosen as the MPN to expand earlier than n_4. It, therefore, becomes insignificant to generate n_4 before expanding n_3. In addition, the above condition also implies that the proof numbers of all the ancestors of node n remain unchanged. As for the disproof numbers of all the ancestors of n, these values are the same or higher. Unfortunately, higher disproof numbers discourage the JL–PNS from choosing n_3 as MPNs to expand. Thus, the behavior becomes awkward, especially if node n_3 will be proved eventually.

One may also ask: what if we expand n_3, but generate n_4 later? In such a case, it may make the proof number of n fluctuate. An extreme situation would be that the value becomes infinity when all nodes, n_1, n_2, and n_3, are disproved, but n is not disproved, since n_4 is not disproved yet.

C. Policies in the Pre-Update Phase

In this section, several policies are proposed for the updates in the pre-update phase. As described in Section II-C, when more workers in the job-level system are available, more MPNs will be selected for execution on these workers. If we do not change the proof/disproof numbers of the chosen MPNs being expanded, named the active MPNs in this paper, we would obviously choose the same node. Therefore, pre-updates are needed to select other nodes as MPNs.

An important goal of choosing multiple MPNs is that these chosen MPNs are also chosen eventually in the case there are no multiple workers, that is, when only one MPN is chosen at a time. Note that the policy without any pre-update is called the native policy in the rest of this paper. Some new policies are introduced and proposed in the subsequent subsections.

1) Virtual Win, Virtual Loss, and Greedy: In this section, we introduce the simplest policies which were also described in the

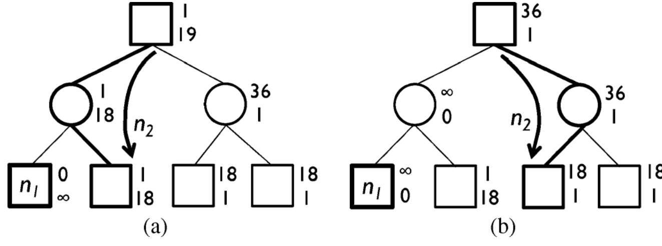


Fig. 6. (a) VW policy. (b) VL policy.

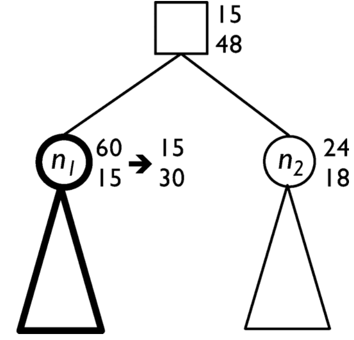


Fig. 7. A starvation example for the VW policy.

preliminary version [31]. One policy, used to prevent choosing the same node twice, named the virtual-win policy (VW policy), assumes a virtual win [8] on the active MPNs. The idea of the VW policy is to assume that the active MPNs are all proved. Thus, their proof/disproof numbers are all set to $0/\infty$, as illustrated in Fig. 6(a). When the proof number of the root is zero, the choosing of more MPNs is stopped, the reason being that the root is already proved if the active MPNs are all proved.

In contrast, another policy, named the virtual-loss policy (VL policy), is to assume a virtual loss on the active MPNs. Thus, the proof/disproof numbers of these nodes are set to $\infty/0$, as shown in Fig. 6(b). Similarly, when the disproof number of the root is zero, we stop choosing more MPNs. Similarly, the root is disproved, if all the active are disproved.

Another introduced policy, named a greedy policy (GD policy), chooses VW policy when the chosen nodes favor a win according to the evaluation of NCTU6, and chooses VL policy otherwise as we may not always be able to decide a winner in advance, as in cases such as the one in Fig. 12(f). The pseudocode for these policies is shown below. The function UpdateAncestors updates the proof/disproof numbers of all the ancestors of the given node n in PNS.

Policy *VirtualWin*(n)
 1: n.pn $= 0$; n, dn $= \infty$;
 2: UpdateAncestors(n);
end policy

Policy *VirtualLoss*(n)
 1: n.pn $= \infty$; n.dn $= 0$;
 2: UpdateAncestors(n);
end policy

Policy *Greedy*(n)
 1: **if** n.pn $\leq n$.dn **then**
 2: n.pn $= 0$; n.dn $= \infty$;
 3: **else**
 4: n.pn $= \infty$; n.dn $= 0$;
 5: **end if**
 6: UpdateAncestors(n);
end policy

As described in Section I, these policies may cause possible fluctuation. From our observation, fluctuation for these policies may result in starvation, as illustrated by the example of VW policy, shown in Fig. 7. In this example, workers are running the jobs for some nodes under the subtree rooted at n_1. Let $p(n_1)/d(n_1)$ be 60/15 for the native policy, and 15/30 for the VW policy. Also, let $p(n_2)/d(n_2)$ be 24/18 for both policies, since no jobs inside the subtree are rooted at n_2.

Now, when a new worker is available, an MPN is chosen for execution. To locate the MPN, the branch to node n_1 is chosen for the VW policy, since $p(n_1) < p(n_2)$. However, for the VW policy, the proof number $p(n_1)$ becomes smaller and the disproof number $d(n_1)$ remains the same or becomes higher according to the MPN property. Subsequently, available workers will continue to choose n_1, as long as jobs remain unfinished. Even if some jobs do finish, the subtree rooted at n_1 will still be chosen as long as $p(n_1)$ remains less than $p(n_2)$. Hence, node n_2 may starve.

The phenomenon of starvation may also happen in both VL and GD policies. In the following sections, further policies are proposed to avoid the above problem.

2) Flag: A simple policy [20] to avoid the above starvation problem, named the flag policy (FG policy) in this paper, is to use a flag mechanism. In this policy, all the MPNs being chosen to generate the first child (like n_3 in Fig. 5) are flagged. Let nodes be called partially flagged nodes, if some of their children are flagged, but others are not, and called fully flagged nodes, if all of their children are flagged. Fully flagged nodes are also flagged recursively. The pseudocode for the FG policy is as follows.

Policy *Flag*(n)
 1: n.flag $= 1$;
 2: **if** all siblings are flagged **then**
 3: Flag(n.parent);
 4: **end if**
end policy

For choosing MPNs, the policy follows the native policy in principle, while avoiding choosing the nodes with flags. Namely, when a chosen node is flagged, another nonflagged node, with the smallest proof/disproof numbers, is chosen instead. An example is illustrated in Fig. 8. In the policy, the next MPN to choose is n_3', since the branch to go from n_1 is n_3.

3) Modified Flag: Although the FG policy can solve the problem of starvation, the example in Fig. 8 shows another potential problem. Node n_1 and all of its ancestors think $p(n_1)$ as well as $p(n_3)$ should be 8. However, the actual value of $p(n_3)$ is 12. In the case that $p(n_3)$ is much larger, the problem is even

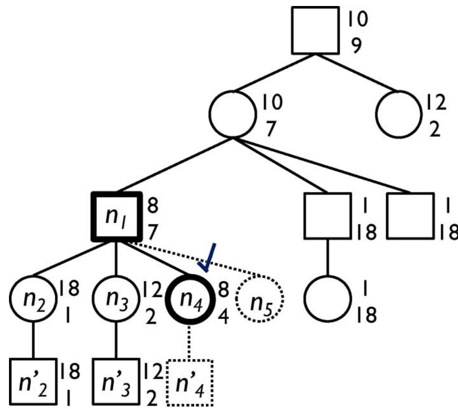


Fig. 8. An example FG policy.

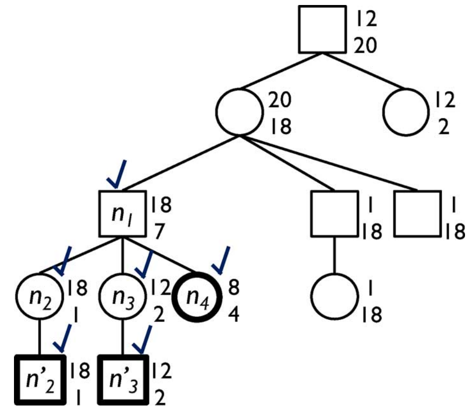


Fig. 9. Assign the maximal proof numbers of children for fully flagged nodes.

more serious. Thus, the policy may lead to the choosing of the wrong MPNs, in this case, n_3.

To solve the above problem, we modify the above policy into a new one, named modified-flag policy (MF policy). The MF policy is as follows. For a partially flagged node, say it is an OR node for simplicity of discussion, its proof number is the minimal proof numbers of nonflagged children. For a fully flagged node, its proof number is the maximum proof number of all flagged children. The pseudocode for MF policy is as follows.

```
Policy ModifiedFlag(n)
  1: Flag(n);
  2: while (n.parent) do
  3:     n = n.parent;
  4:     if n is an OR node then
  5:         n.dn = sum(c.dn) for all children c;
  6:         if n has nonflagged children then
  7:             n.pn = min(c.pn) for all nonflagged
                 children c;
  8:         else
  9:             n.pn = max(c.pn) for all flagged
                 children c;
  10:        end if
  11:    else
  12:        //omitted
  13:    end if
  14: end while
end policy
```

For a fully flagged node, we set its proof number to the maximum proof number among children, instead of the minimum one. The reasoning behind this is illustrated by the example in Fig. 8. Assume that for node n_1, the two children n_3 and n_4 are flagged and child n_2 is not flagged yet. The value $p(n_1)$ is 18. Now, we look to select one more MPN from n_1. Node n'_2 is then selected. According to the FG policy, where the proof number is set to the minimum, value $p(n_1)$ then drops to 8. This implies that the next MPN selection will be attracted toward node n_1. This is clearly awkward. In the case that we set the proof number to the maximum proof number among children, value $p(n_1)$ remains 18. Thus, this policy does not wrongly direct the MPN selection.

Fig. 9 shows the proof/disproof numbers of search trees in Fig. 8 in the MF policy. As for disproof numbers (in OR nodes) for the above case, we still follow the PNS to sum up the disproof numbers of all the children, regardless of whether they are flagged. For example, in both Figs. 8 and 9, $d(n_1)$ is 7.

4) Virtual Equivalence: VE is an idea based on the assumption that the generated node is expected to have almost the same proof/disproof numbers as its parent, if the generated node is the eldest child, or as the youngest elder sibling, otherwise. The pseudocode for this policy is as follows.

```
Policy VirtualEquivalence(n)
  1: if n has sibling then
  2:     set s to the youngest elder sibling;
  3: else
  4:     set s to the parent;
  5: end if
  6: n.pn = s.pn;
  7: n.dn = s.dn;
  8: UpdateAncestors(n);
end policy
```

The following two cases are discussed for this policy. First, assume that node n has no child yet. Then, when a program like NCTU6 is used to generate from n (regardless of the AND/OR node) the first node n_1, the best move from n, it is expected that the calculated value which is used to initialize the proof/disproof values of n_1 is the same as or close to that for n, based on the assumption that the program is accurate enough.

Second, assume that node n has some children, say three, n_1, n_2, and n_3, generated based on the scheme of the postponed sibling generations. As per the argument in the postponed sibling generations, the three children stand for the best, the second best, and the third best children of node n, respectively. Now, when the program is to generate the fourth child n_4, that is, the fourth best child, it is expected that the calculated value for n_4 is the same as or close to that for n_3.

In fact, the FG policy can be viewed as a kind of the first case. For example, in Fig. 8, the generation of a new node n'_4 is assumed, whose proof/disproof numbers of n'_4 are the same as those of n_4.

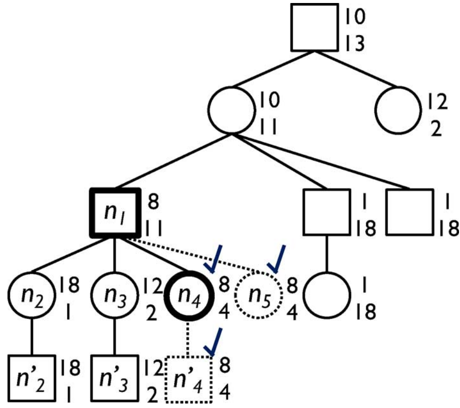


Fig. 10. The search tree in Fig. 9 with the FG–VE policy.

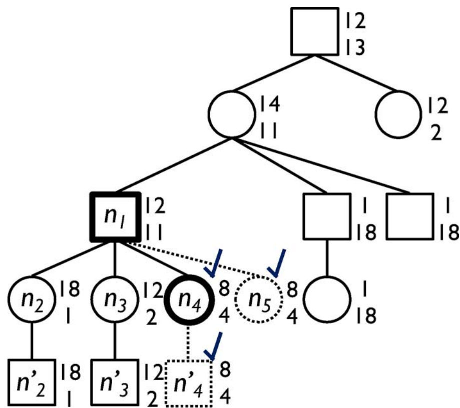


Fig. 11. The search tree in Fig. 9 with the MF–VE policy.

Now, let us investigate the second case. For this argument, we look to generate a new child whose proof/disproof numbers are the same as those of its youngest elder sibling, i.e., for the example in Fig. 8, we set the proof/disproof numbers of n_5 to those of node n_4.

Based on the discussion above, both FG and MF policies can be modified into flag-with-virtual-equivalence policy (FG–VE policy) and modified-flag-with-virtual-equivalence policy (MF–VE policy), respectively. Both Figs. 10 and 11 show proof/disproof numbers of the PNS tree in Fig. 9 for both FG–VE and MF–VE policies, respectively.

V. Experiments

In our experiments, our job-level system is maintained on a desktop grid [30] with eight workers, Intel Core2 Duo 3.33-GHz machine. Since each worker has two cores, the desktop has actually 16 cores in total. And, the client was located on another host. Note that the time for maintaining the JL–PNS tree in the client is negligible, since it is relatively low when compared with that for NCTU6.

In our experiments of JL–PNS, the benchmark included 35 *Connect6* positions (available in [38], and also the same as those in the preliminary version [31]), among which the last 15 positions are won by the player-to-move, while the first 20 positions are won by the other player. The first 20 and the last 15 positions are ordered according to their computation times on one core.

Among the 35 positions, ten are three-move openings shown in Fig. 12(a)–(j). For many of them, their winning strategies had not been found before our work. In particular, the Mickey Mouse opening [Fig. 12(i)] had been one of the most popular openings before we solved it. Fig. 13 shows a path in the winning tree. The tenth one [Fig. 12(j)], also called straight opening, is another difficult one.

According to our statistics on running the 35 positions, each NCTU6 job takes about 37.45 s on average. About 21.10% of the jobs are run over 1 min. About 14.99% of jobs are returned with wins/losses, and these jobs are usually run quickly. If these jobs are not counted, each NCTU6 job takes about 41.38 s on average. In addition, 11.19% extra jobs are aborted.

In this section, for performance analysis, let speedup S_k be T_1/T_k, where T_k is the computation time for solving a position with k cores. Also, let efficiency E_k be S_k/k. The efficiencies are one for ideal linear speedups.

This section is organized as follows. Section V-A details the experiments for our benchmark, comparing all the polices mentioned in Section IV-C. The results show that the four policies, FG, MF, FG–VE, and MF–VE, are clearly better than the other three. Section V-B discusses the accuracy of VE by showing status correlations between nodes and their parents or sibling nodes. Then, we further analyze the experimental results of the four policies in Section V-C. In Section V-D, we analyze the performances for the positions requiring more computation times.

A. Experiments for Benchmark

We performed experiments for our benchmark to investigate all the policies mentioned in Section IV-C. For each of these policies, we measured their computation times with 1, 2, 4, 8, and 16 cores for each *Connect6* position.

In order to have a quick comparison and performance analysis, we compared the efficiencies of the 35 *Connect6* positions, for each policy and for each number of cores, as shown in Fig. 14. Note that all the one-core performance results for the different policies are the same since there are no differences in choosing nodes on a single core for different policies.

From Fig. 14, the four policies with the flag mechanism (FG, MF, FG–VE, and MF–VE) outperformed the other three without flag mechanism (VW, VL, and GD). For example, the computation times for VW, VL, or GD with 16 cores were about 80% longer than those of MF–VE. From our observation, we did find some cases with starvation phenomenon, as mentioned in Section IV-C1.

The performances for the policies with the flag mechanism are close and will be discussed in more detail in Section V-C. Before discussing these, we give an analysis of VE in Section V-B.

B. The Analysis for VE

In Section IV-C4, the concept of VE is as follows: the generated node is expected to have almost the same proof/disproof numbers as its parent, and as the youngest elder sibling. In this section, our experiments are designed to test how close they are. For example, how close are the $p(n)/d(n)$ of the two generating nodes, n_4 and n'_4, and/or n_4 and n_5 in Fig. 11? To assess this,

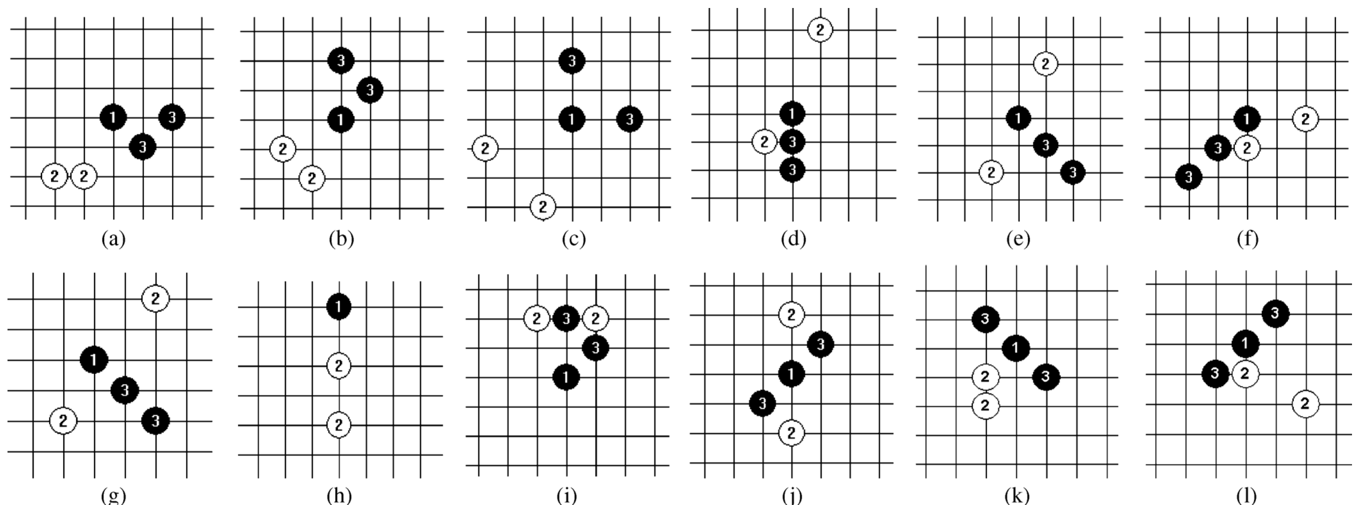


Fig. 12. The 12 solved openings.

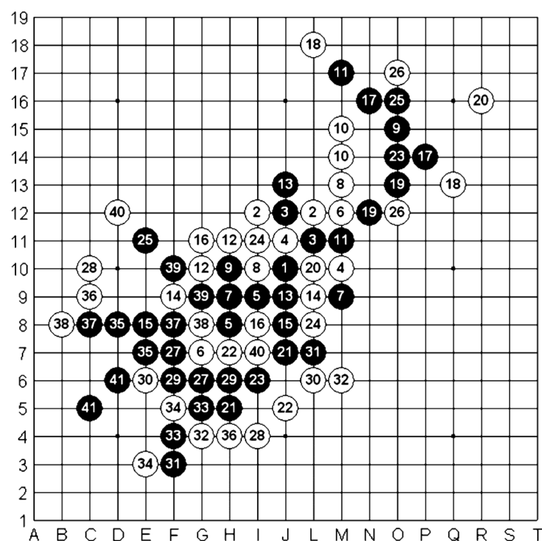


Fig. 13. A path in the winning tree of the Mickey Mouse opening.

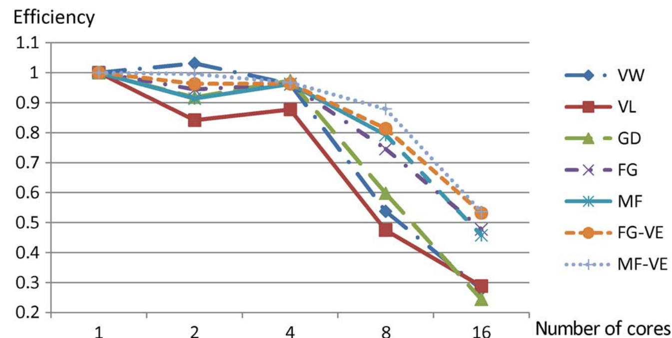


Fig. 14. The efficiencies for all 35 positions for each policy.

we measure the distances between n_4 and n'_4, and between n_4 and n_5.

For the measurement, we assign a value to each status, as shown in Table II, and calculate the distances. For example, in Fig. 11, the distance between n_3 and n_4 is 8 minus 6, since W3 is 8 and W1 is 6. Notably, for "unstable" positions, since they are hard to locate, we simply ignore the distances with the unstable

TABLE II
VALUE ASSIGNED FOR EACH STATUS

Status	B:w	B4	B3	B2	B1	stable
v(status)	0	1	2	3	4	5
Status	W:w	W4	W3	W2	W1	unstable
v(status)	10	9	8	7	6	*

status. The procedure of counting the distance is as follows.

Procedure *Count Distance*(n)
1: **if** n is eldest child **then**
2: p = parent of n;
3: $par_dist += \mathrm{v}(n.status) - \mathrm{v}(p.status)$;
4: **else**
5: s = youngest elder sibling of n;
6: $sib_dist += \mathrm{v}(n.status) - \mathrm{v}(s.status)$;
7: **end if**
end

For all nodes generated in solving the 35 positions, we show the statistics for the distances between neighboring siblings and between parents and the eldest children in Fig. 15. As seen in the figure, most generated nodes have the same status (the distances are 0) as the eldest child and as the eldest younger sibling. According to this result, it is expected that the proof/disproof numbers will be less fluctuated. Thus, it becomes more likely that the chosen MPNs will also be chosen in the single core version.

From Fig. 15, we also observe that the distances between parents and the eldest children are, in general, larger than those between siblings. The reason for this is similar to the two-ply update issue, mentioned in [36]. Since a parent has two fewer stones than its children in *Connect6*, it is harder to evaluate them consistently.

C. Flag Mechanism

This section analyzes the performances of the policies with the flag mechanism in more detail. Fig. 16 shows the ratios of the performances of different versions with respect to those of the FG. In this figure, we observe that the MF–VE policy performed

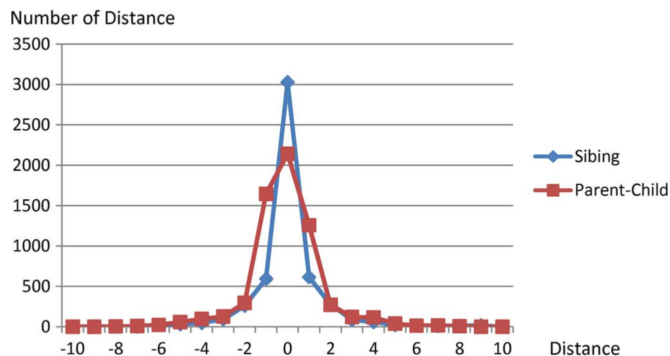


Fig. 15. Measurement of each distance.

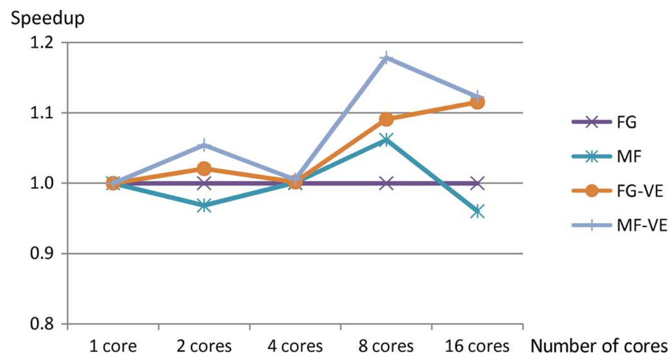


Fig. 16. The speedups relative to the FG policy for solving 35 *Connect6* positions with different policies.

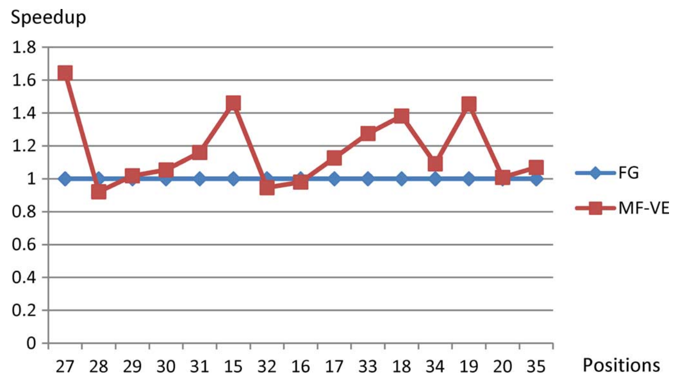


Fig. 17. The improvement of the speedup for the most difficult 15 positions from FG to MF–VE.

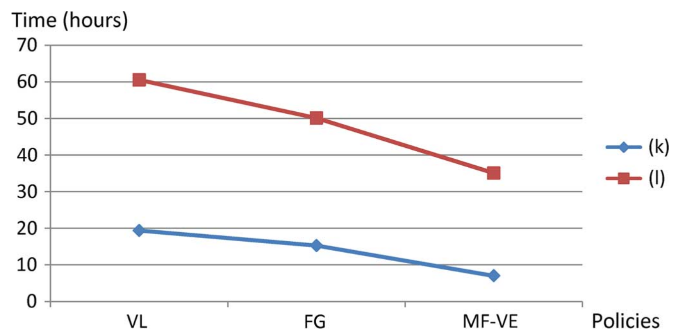


Fig. 18. The solving times for the three versions of each position on 16 cores.

best and outperformed FG by about 17.8% and 12.3% for 8 cores and 16 cores, respectively.

Fig. 16 also shows that the performances of both FG–VE and MF–VE are better than those with MF and FG. Both FG–VE and MF–VE use the sibling VE, while the other two do not. This indicates that the policies with the sibling VE perform better.

D. Experiments for Difficult Positions

In this section, we analyze performance for the positions requiring more computation. For this purpose, we chose 15 of the most difficult positions among the 35, and analyzed the improvements from FG to MF–VE using 16 cores, as shown in Fig. 17. Note that the positions in Fig. 17 are ordered according to the computation time, with the rightmost one requiring the most time. This is about 2.75 h for 16 cores. From this figure, we observe that MF–VE generally performed slightly better than FG.

Thereafter, we investigated some other positions requiring even more computation times. After our preliminary work in [31], we solved two more openings, as shown in Fig. 12(k) and (l). These required significantly more computation time, about 7.03 and 35.11 h for 16 cores, respectively. Since much more time was spent in solving the two openings, we only compared three policies (VL, FG, and MF–VE) by running them on 16 cores. The computation times are shown in Fig. 18.

As seen in Fig. 18, MF–VE performed better than FG by factors of 2.18 and 1.43 for the positions in Fig. 12(k) and (l), respectively. Moreover, MF–VE performed better than VL by 2.76 and 1.72 for Fig. 12(k) and (l), respectively. The results demonstrate that MF–VE also outperforms FG in bigger cases.

VI. Discussion

In this section, we first discuss some past job-level-like research work, and then some issues about job-level computation in the implementations.

A. Past Job-Level-Like Work

The research into solving *Checkers* [22] was separated into two parts: the proof-tree manager and the proof solvers. The proof-tree manager, like the client in our model, used the PNS to identify a prioritized list of positions to be examined by the proof solvers, like jobs in our model. Their manager generated several hundred positions at a time to keep workers busy, and they did not consider the pre-update.

Chaslot *et al.* [7] proposed a meta MCTS to build openings in the book of *Go*. In their method, a tree policy was used to select a node in the upper confidence tree (UCT) [15], while an MCTS program was used to generate moves in the simulation. The program maintaining the UCT tree acts as the client, while the MCTS program used in the simulation acts as the job.

We believe that the job-level computation model can also be easily applied to many other search techniques. In addition to JL–PNS, our ongoing projects are seeking to apply the job-level model to other game search applications [29], such as job-level Monte Carlo tree search (JL–MCTS) for *Go* and job-level alpha–beta search (JL–ABS) for *Chinese Chess*.

B. Miscellaneous Issues

The first issue to be discussed is that of the overhead of job dispatching. In the job-level model, the client must wait passively for notification of idling workers. Thus, the overhead is incurred for the round trip of notification. In practice, in the job-level system [30], one or more jobs are dispatched to the broker in advance, to keep all workers busy. Note that we do not dispatch a large number of jobs in advance for the reason mentioned in Section II-A.

The second issue is that of distributed versus shared memory. One key for our job-level model is to leverage a game-playing program which can be encapsulated as a job to be dispatched to a worker remotely in a distributed computing environment. Distribution, however, means that some data, such as transposition tables, cannot be shared by different jobs. However, if the job supports several threads and the worker offers several cores, then the job can still be run with several threads on the worker in the job-level system.

The third discussion is that of the quality of game-playing programs. In our experiences of using JL–PNS, we observed that the quality of game-playing programs affects the total computation time significantly. In our earlier versions of NCTU6, we could not solve the straight opening after 100 000 jobs, and solved the Mickey Mouse opening with many more than that. After we improved NCTU6 in later versions, the straight opening, as well as many other positions, was solved, and the Mickey Mouse opening was solved with fewer jobs (almost half). On the other hand, JL–PNS or job-level search can be used to indicate the quality of game-playing programs.

VII. Conclusion

This paper introduces an approach, generic job-level search, to leverage game-playing programs which are already written and encapsulated as jobs. In this paper, we present and focus on JL–PNS, a kind of generic job-level search, and apply JL–PNS to automatically solving several *Connect6* positions, including some difficult openings. The contributions of this paper are summarized as follows.

- This paper proposes the job-level computation model. As described in Section I, the benefits of job level include the following: develop clients and jobs independently, run jobs in parallel, maintain the generic search in the client, and monitor the search tree easily. The first also implies that it is easy to develop job-level search without extra developmental cost to the game-playing programs (like NCTU6).
- This paper proposes a new approach, JL–PNS, to help solve the openings of *Connect6*.
- This paper successfully uses JL–PNS to solve several positions of *Connect6* automatically, including several three-move openings in Fig. 12. No *Connect6* human experts were able to solve them. From the results, we expect to solve and develop more *Connect6* openings.
- For JL–PNS, this paper proposes some techniques, such as the method of postponed sibling generation and the policies of choosing MPNs.
- Our experiments demonstrated that the MF–VE policy performs best. Thus, it is recommended to use this policy to solve positions.
- Our experiments demonstrated an average speedup of 8.58 on 16 cores.

In addition to JL–PNS, our future work will be to apply the job-level model to other applications [29], such as JL–MCTS for *Go* and JL–ABS for *Chinese Chess*.

Acknowledgment

The authors would like to thank O. Teytaud and the anonymous referees for their valuable comments, the National Center for High-performance Computing (NCHC) for computer time and facilities, and Chunghwa Telecom for computer time and facilities of HiCloud.

References

[1] L. V. Allis, "Searching for solutions in games and artificial intelligence," Ph.D. dissertation, Dept. Comput. Sci., Univ. Limburg, Maastricht, The Netherlands, 1994.
[2] L. V. Allis, H. J. van den Herik, and M. P. H. Huntjens, "Go-Moku solved by new search techniques," *Comput. Intell.*, vol. 12, pp. 7–23, 1996.
[3] L. V. Allis, M. van der Meulen, and H. J. van den Herik, "Proof number search," *Artif. Intell.*, vol. 66, no. 1, pp. 91–124, 1994.
[4] D. P. Anderson, "Boinc: A system for public-resource computing and storage," in *Proc. 5th IEEE/ACM Int. Workshop Grid Comput.*, Pittsburgh, PA, USA, 2004, pp. 4–10.
[5] D. M. Breuker, J. Uiterwijk, and H. J. van den Herik, "The PN^2-search algorithm," in *Advances in Computer Games*, H. J. van den Herik and B. Monien, Eds. Maastricht, The Netherlands: IKAT, Universiteit Maastricht, 2001, vol. 9, pp. 115–132.
[6] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of Monte Carlo tree search method," *IEEE Trans. Comput. Intell. AI Games*, vol. 4, no. 1, pp. 1–43, Mar. 2012.
[7] G. M. Chaslot, J.-B. Hoock, J. Perez, A. Rimmel, O. Teytaud, and M. H. M. Winands, "Meta Monte-Carlo tree search for automatic opening book generation," in *Proc. Workshop General Intell. Game Playing Agents*, 2009, pp. 7–12.
[8] G. M. Chaslot, M. H. M. Winands, and H. J. van den Herik, "Parallel Monte-Carlo tree search," in *Proc. 6th Int. Conf. Comput. Games*, Beijing, China, 2008, pp. 60–71.
[9] G. Fedak, C. Germain, V. Neri, and F. Cappello, "Xtremweb: A generic global computing system," in *Proc. 1st IEEE/ACM Int. Symp. Cluster Comput. Grid: Workshop Global Comput. Pers. Devices*, Brisbane, Australia, 2001, pp. 582–587.
[10] S. Gelly and D. Silver, "Monte-Carlo tree search and rapid action value estimation in computer Go," *Artif. Intell.*, vol. 175, pp. 1856–1875, Jul. 2011.
[11] H. J. van den Herik and M. H. M. Winands, "Proof number search and its variants," in *Oppositional Concepts in Computational Intelligence*, ser. Studies in Computational Intelligence. New York, NY, USA: Springer-Verlag, 2008, vol. 155, pp. 91–118.
[12] H. J. van den Herik, J. W. H. M. Uiterwijk, and J. V. Rijswijck, "Games solved: Now and in the future," *Artif. Intell.*, vol. 134, pp. 277–311, 2002.
[13] A. Kishimoto and Y. Kotani, "Parallel AND/OR tree search based on proof and disproof numbers," in *Proc. 5th Games Programming Workshop*, 1999, vol. 99, no. 14, pp. 24–30.
[14] A. Kishimoto and M. Müller, "DF-PN in Go: Application to the One-Eye Problem," in *Advances in Computer Games Conference (ACG'10)*, H. J. van den Herik, H. Iida, and E. A. Heinz, Eds. Norwell, MA, USA: Kluwer, 2003, pp. 125–141.
[15] L. Kocsis and C. Szepesvari, "Bandit-based Monte-Carlo planning," in *European Conference on Machine Learning (ECML'06)*, ser. Lecture Notes in Computer Science. Berlin, Germany: Springer-Verlag, 2006, vol. 4212, pp. 282–293.

[16] H.-H. Lin, D.-J. Sun, I.-C. Wu, and S.-J Yen, "The 2010 TAAI computer-game tournaments," *Int. Comput. Games Assoc. J.*, vol. 34, no. 1, pp. 51–54, Mar. 2011.

[17] P.-H. Lin and I.-C. Wu, "NCTU6 wins in the man-machine Connect6 championship 2009," *Int. Comput. Games Assoc. J.*, vol. 32, no. 4, pp. 230–232, 2009.

[18] A. Nagai, "DF-PN algorithm for searching AND/OR trees and its applications," Ph.D. dissertation, Dept. Inf. Sci., Univ. Tokyo, Tokyo, Japan, 2002.

[19] J. Pawlewicz and L. Lew, "Improving depth-first PN-search: $1+\varepsilon$ trick," in *5th International Conference on Computers and Games*, ser. Lecture Notes in Computer Science, H. J. van den Herik, P. Ciancarini, and H. H. L. M. Donkers, Eds. Berlin, Germany: Springer-Verlag, 2006, vol. 4630, pp. 160–170.

[20] A. Saffidine, N. Jouandeau, and T. Cazenave, "Solving breakthrough with race patterns and job-level proof number search," in *Proc. 13th Adv. Comput. Games Conf.*, Tilburg, The Netherlands, 2011, pp. 196–207.

[21] J. T. Saito, M. H. M. Winands, and H. J. van den Herik, "Randomized parallel proof number search," in *Advances in Computer Games Conference (ACG'12)*, ser. Lecture Notes in Computer Science. Berlin, Germany: Springer-Verlag, 2009, vol. 6048, pp. 75–87.

[22] J. Schaeffer, N. Burch, Y. N. Björnsson, A. Kishimoto, M. Müller, R. Lake, P. Lu, and S. Sutphen, "Checkers is solved," *Science*, vol. 5844, no. 317, pp. 1518–1552, 2007.

[23] M. Seo, H. Iida, and J. Uiterwijk, "The PN*-search algorithm: Application to Tsumeshogi," *Artif. Intell.*, vol. 129, no. 1–2, pp. 253–277, 2001.

[24] SETI@home Project [Online]. Available: http://setiathome.ssl.berkeley.edu

[25] Taiwan Connect6 Association [Online]. Available: http://www.connect6.org/

[26] TCGA Association, "TCGA Computer Game Tournaments," [Online]. Available: http://tcga.ndhu.edu.tw/TCGA2011/

[27] T. Thomsen, "Lambda-search in game trees—With application to Go," *Int. Comput. Games Assoc. J.*, vol. 23, no. 4, pp. 203–217, 2000.

[28] M. H. M. Winands, J. W. H. M. Uiterwijk, and H. J. van den Herik, "PDS-PN: A new proof number search algorithm: Application to lines of action," in *Computers and Games 2002*, ser. Lecture Notes in Computer Games, J. Schaeffer, M. Müller, and Y. Björnson, Eds. Berlin, Germany: Springer-Verlag, 2003, vol. 2883, pp. 170–185.

[29] I.-C. Wu, S.-C. Hsu, S.-J. Yen, S.-S. Lin, K.-Y. Kao, J.-C. Chen, K.-C. Huang, H.-Y. Chang, and Y.-C. Chung, "A volunteer computing system for computer games and its applications," National Science Council, Taiwan, Integrated Project, 2010.

[30] I.-C. Wu, C.-P. Chen, P.-H. Lin, K.-C. Huang, L.-P. Chen, D.-J. Sun, Y.-C. Chan, and H.-Y. Hsou, "A volunteer-computing-based grid environment for Connect6 applications," in *Proc. 12th IEEE Int. Conf. Comput. Sci. Eng.*, Vancouver, BC, Canada, Aug. 29–31, 2009, pp. 110–117.

[31] I.-C. Wu, H.-H. Lin, P.-H. Lin, D.-J. Sun, Y.-C. Chan, and B.-T. Chen, "Job-level proof number search for Connect6," in *Proc. Int. Conf. Comput. Games*, Kanazawa, Japan, 2010, pp. 11–22.

[32] I.-C. Wu, D.-Y. Huang, and H.-C. Chang, "Connect6," *Int. Comput. Games Assoc. J.*, vol. 28, no. 4, pp. 234–242, 2006.

[33] I.-C. Wu and D.-Y. Huang, "A new family of k-in-a-row games," in *Proc. 11th Adv. Comput. Games Conf.*, Taipei, Taiwan, 2005, pp. 180–194.

[34] I.-C. Wu and P.-H. Lin, "NCTU6-lite wins Connect6 tournament," *Int. Comput. Games Assoc. J.*, vol. 31, no. 4, pp. 240–243, 2008.

[35] I.-C. Wu and P.-H. Lin, "Relevance-zone-oriented proof search for Connect6," *IEEE Trans. Comput. Intell. AI Games*, vol. 2, no. 3, pp. 191–207, Sep. 2010.

[36] I.-C. Wu, H.-T. Tsai, H.-H. Lin, Y.-S. Lin, C.-M. Chang, and P.-H. Lin, "Temporal difference learning for Connect6," in *Advances in Computer Games (ACG 13)*, ser. Lecture Notes in Computer Science. Berlin, Germany: Springer-Verlag, 2012, vol. 7168, pp. 121–133.

[37] I.-C. Wu and S.-J. Yen, "NCTU6 wins Connect6 tournament," *Int. Comput. Games Assoc. J.*, vol. 29, no. 3, pp. 157–158, Sep. 2006.

[38] I.-C. Wu, H.-H. Lin, D.-J. Sun, K.-Y. Kao, P.-H. Lin, Y.-C. Chan, and P.-T. Chen, "Benchmark for Connect6," [Online]. Available: http://www.connect6.org/articles/JL-PNS_2012/

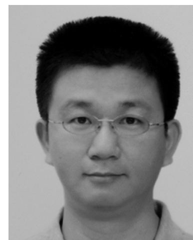
I.-Chen Wu (M'10) received the B.S. degree in electronic engineering and the M.S. degree in computer science from the National Taiwan University (NTU), Taipei, Taiwan, in 1982 and 1984, respectively, and the Ph.D. degree in computer science from Carnegie Mellon University, Pittsburgh, PA, USA, in 1993.

He is with the Department of Computer Science, National Chiao Tung University, Hsinchu, Taiwan. His research interests include artificial intelligence, Internet gaming, volunteer computing, and cloud computing.

Dr. Wu introduced a new game *Connect6*, a kind of six-in-a-row game, and presented this game at the 11th Advances in Computer Games Conference (ACG) in 2005. Since then, *Connect6* has become a tournament item in Computer Olympiad. He led a team developing a *Connect6* program, named NCTU6. The program won the gold twice in Computer Olympiad in both 2006 and 2008.

Hung-Hsuan Lin received the B.S. degree from the Department of Computer Science, National Chiao Tung University, Hsinchu, Taiwan, where he is currently working toward the Ph.D. degree in computer science.

His research interests include artificial intelligence, computer game, volunteer computing, and cloud computing.



Der-Johng Sun is currently working toward the Ph.D. degree in computer science in the Department of Computer Science, National Chiao Tung University, Hsinchu, Taiwan.

His research interests include artificial intelligence and grid and cloud computing.

Kuo-Yuan Kao received the Ph.D. degree in mathematics from the University of North Carolina at Charlotte, Charlotte, NC, USA, in 1997.

His program won the U.S. Computer Go Championship in 1994. Since 1995, he has engaged in research of combinatorial game theory and published several papers in this field. Kao is also a 6-dan amateur Go player. He currently serves as the Director of the Chinese Go Association, Taiwan.

Ping-Hung Lin received the Ph.D. degree in computer science from the National Chiao Tung University, Hsinchu, Taiwan, in 2010.

He is an Engineer in the Industrial Technology Research Institute, Hsinchu, Taiwan. He is one of the major designers of the *Connect6* program NCTU6 that won the gold two times in Computer Olympiad in 2006 and 2008. His research interests include artificial intelligence and cloud computing.

Yi-Chih Chan received the M.S. degree in computer science from the National Chiao Tung University, Hsinchu, Taiwan, in 2009.

His research interests include artificial intelligence and grid and cloud computing.

Po-Ting Chen received the M.S. degree in computer science from the National Chiao Tung University, Hsinchu, Taiwan, in 2010.

His research interests include artificial intelligence and cloud computing.